



Министерство науки и высшего образования Российской Федерации

**Федеральное государственное бюджетное образовательное учреждение высшего образования
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ имени Н.Э.БАУМАНА
(национальный исследовательский университет)»**

Факультет: Информатика и системы управления

Кафедра: Теоретическая информатика и компьютерные технологии

Лабораторная работа № 4

Объектно-ориентированный лексический анализатор
по дисциплине «Конструирование компиляторов»

Вариант 27

Работу выполнил
студент группы ИУ9-62Б
Жук Дмитрий

Москва, 2022

Цель работы

Целью данной работы является приобретение навыка реализации лексического анализатора на объектно-ориентированном языке без применения каких-либо средств автоматизации решения задачи лексического анализатора.

Задание

В лабораторной работе предлагается реализовать две первые фазы стадии анализа: чтение входного потока и лексический анализ. При этом следует придерживаться схемы реализации объектно-ориентированного лексического анализатора, рассмотренной на лекции.

Для лексических доменов должны вычисляться их атрибуты:

- для целых чисел атрибут должен быть целым числом;
- для идентификаторов – номер в таблице идентификаторов;
- для строковых констант – значение, изображаемое самой строковой константой (т.е. без окружающих кавычек и с интерпретацией escape-последовательностей)

Индивидуальный вариант

Символьные литералы: ограничены апострофами, могут содержать Escape-последовательности «\'», «\n», «\\» и «\xxxx» (здесь буквы «x» обозначают шестнадцатеричные цифры). Идентификаторы: последовательности буквенно-цифровых символов Unicode длиной от 2 до 10 символов, начинающиеся и заканчивающиеся буквой. Ключевые слова: «z», «for», «forward».

Реализация

```
1: use colored::Colorize;
2: use std::collections::HashMap;
3: use std::env;
4: use std::vec::Vec;
5:
6: #[derive(Debug)]
7: enum TokenType {
8:     Spaces(String),
9:     Symbol(char),
10:    Identifier(usize),
11:    Key(String),
12:    Error(char),
13:    SimpleError,
14:    End,
15: }
16:
17: #[derive(Debug)]
18: struct Token {
19:     from: (usize, usize),
20:     to: (usize, usize),
21:     value: TokenType,
22: }
23:
24: #[derive(Debug)]
25: struct SmartIterator {
26:     arr: Vec<char>,
27:     prev_pos: (usize, usize),
28:     pos: (usize, usize),
29:     ind: usize,
30: }
31:
32: #[derive(Debug)]
33: struct Position {
34:     ind: usize,
35:     pos: (usize, usize),
36:     prev_pos: (usize, usize),
37: }
38:
39: impl SmartIterator {
40:     fn new(s: String) -> Self {
41:         Self {
42:             arr: s.chars().collect(),
43:             pos: (1, 1),
44:             prev_pos: (1, 0),
45:             ind: 0,
46:         }
47:     }
48:
49:     fn see(&self) -> Option<char> {
50:         if self.ind < self.arr.len() {
51:             Some(self.arr[self.ind])
52:         } else {
53:             None
54:         }
55:     }
56: }
```

```

54:     }
55: }
56:
57: fn next(&mut self) -> Option<char> {
58:     if self.ind < self.arr.len() {
59:         Some({
60:             let x = self.arr[self.ind];
61:             self.ind += 1;
62:             self.prev_pos = self.pos;
63:             self.pos = if x == '\n' {
64:                 (self.pos.0 + 1, 1)
65:             } else {
66:                 (self.pos.0, self.pos.1 + 1)
67:             };
68:             x
69:         })
70:     } else {
71:         None
72:     }
73: }
74:
75: fn save_pos(&self) -> Position {
76:     Position {
77:         ind: self.ind,
78:         pos: self.pos,
79:         prev_pos: self.prev_pos,
80:     }
81: }
82:
83: fn load_pos(&mut self, a: Position) {
84:     self.ind = a.ind;
85:     self.pos = a.pos;
86:     self.prev_pos = a.prev_pos;
87: }
88: }
89:
90: #[derive(Debug)]
91: struct ParseToken(SmartIterator, HashMap<String, usize>);
92:
93: fn to_digit_16(d: char) -> Option<u32> {
94:     d.to_digit(16)
95: }
96:
97: impl ParseToken {
98:     fn new(iter: SmartIterator) -> Self {
99:         Self(iter, HashMap::new())
100:     }
101:
102:     fn next_spaces(&mut self) -> TokenType {
103:         let mut ans = String::new();
104:         while let Some(s @ (' ' | '\t' | '\n')) = self.0.next() {
105:             ans.push(s);
106:             self.0.next();
107:         }
108:         TokenType::Spaces(ans)
109:     }
110: }

```

```

111:   fn next_number16_4(&mut self, x: char) -> Option<char> {
112:       to_digit_16(x)
113:       .zip(self.0.next().and_then(to_digit_16))
114:       .zip(self.0.next().and_then(to_digit_16))
115:       .zip(self.0.next().and_then(to_digit_16))
116:       .and_then(|((x1, x2), x3), x4)| char::from_u32(((x1 * 16 + x2) *
16 + x3) * 16 + x4))
117:   }
118:
119:   fn next_symbol(&mut self) -> TokenType {
120:       if Some('\\\'') == self.0.next() {
121:           if let Some(ans) = match self.0.next() {
122:               Some('\\\'') => None,
123:               Some('\\n') => None,
124:               Some('\\\'') => match self.0.next() {
125:                   Some('n') => Some('\\n'),
126:                   Some(x @ ('\\\' | '\\\'')) => Some(x),
127:                   Some(x1) => self.next_number16_4(x1),
128:                   _ => None,
129:               },
130:               x @ _ => x,
131:           } {
132:               if Some('\\\'') == self.0.next() {
133:                   TokenType::Symbol(ans)
134:               } else {
135:                   TokenType::SimpleError
136:               }
137:           } else {
138:               TokenType::SimpleError
139:           }
140:       } else {
141:           TokenType::SimpleError
142:       }
143:   }
144:
145:   fn next_identifier_or_key(&mut self) -> TokenType {
146:       if let Some(x) = self.0.next() {
147:           if x.is_alphabetic() {
148:               let mut ans = String::from(x);
149:               let mut last_true_save = Some((self.0.save_pos(), ans.clone()));
150:               loop {
151:                   match self.0.see() {
152:                       Some(x) if x.is_alphabetic() || x.is_digit(10) => {
153:                           ans.push(x);
154:                           self.0.next();
155:                           if x.is_alphabetic() {
156:                               last_true_save = Some((self.0.save_pos(), ans.clone()))
157:                           }
158:                       }
159:                       _ => break,
160:                   }
161:                   if ans.len() == 10 {
162:                       break;
163:                   }
164:               }
165:               match last_true_save {

```

```

166:         Some((save, ans)) if ans.eq("z") || ans.eq("for") ||
ans.eq("forward") => {
167:             self.0.load_pos(save);
168:             TokenType::Key(ans)
169:         }
170:         Some((save, ans)) if ans.len() >= 2 => {
171:             self.0.load_pos(save);
172:             let len = self.1.len();
173:             TokenType::Identifier(*self.1.entry(ans).or_insert(len))
174:         }
175:         _ => TokenType::SimpleError,
176:     } else {
177:         TokenType::SimpleError
178:     } else {
179:         TokenType::SimpleError
180:     } else {
181:         TokenType::SimpleError
182:     }
183: }
184: }
185:
186: impl std::iter::Iterator for ParseToken {
187:     type Item = Token;
188:
189:     fn next(&mut self) -> Option<Token> {
190:         if let Some(l) = self.0.see() {
191:             let save = self.0.save_pos();
192:             let token = match l {
193:                 ' ' | '\t' | '\n' => self.next_spaces(),
194:                 '\\' => self.next_symbol(),
195:                 x if x.is_alphabetic() => self.next_identifier_or_key(),
196:                 _ => TokenType::SimpleError,
197:             };
198:             match token {
199:                 TokenType::SimpleError => {
200:                     self.0.load_pos(save);
201:                     if env::var("SKIP_ERRORS").is_ok() {
202:                         self.0.next();
203:                         self.next()
204:                     } else {
205:                         Some(Token {
206:                             from: self.0.pos,
207:                             value: TokenType::Error(self.0.next().unwrap()),
208:                             to: self.0.pos,
209:                         })
210:                     }
211:                 }
212:                 TokenType::Spaces(_) if !env::var("NEED_SPACES").is_ok() =>
self.next(),
213:                 _ => Some(Token {
214:                     from: save.pos,
215:                     value: token,
216:                     to: self.0.prev_pos,
217:                 }),
218:             }
219:         } else if !env::var("SKIP_EOF").is_ok() {
220:             self.0.ind += 1;

```

```

221:         Some(Token {
222:             from: self.0.pos,
223:             value: TokenType::End,
224:             to: self.0.pos,
225:         })
226:     } else {
227:         None
228:     }
229: }
230: }
231:
232: fn main() {
233:     let filename = std::env::args().nth(1).unwrap();
234:
235:     println!("filename = {:?}", filename);
236:
237:     let content = std::fs::read_to_string(filename).unwrap();
238:
239:     println!("content = {:?}", content);
240:     println!();
241:
242:     let mut iter = ParseToken::new(SmartIterator::new(content));
243:     loop {
244:         match iter.next() {
245:             Some(Token {
246:                 value: TokenType::End,
247:                 ..
248:             }) => {
249:                 println!("HashMap IDN:");
250:                 for (key, value) in &iter.1 {
251:                     println!("{:<20} : {}", key.blue(), format!("{:?}",
value).blue());
252:                 }
253:                 break;
254:             }
255:             Some(x) => {
256:                 let (name, val) = match x.value {
257:                     TokenType::Spaces(sp) => ("SPA".white(), format!("{:?}",
sp).white()),
258:                     TokenType::Symbol(str) => ("SYM".green(), format!("{:?}",
str).green()),
259:                     TokenType::Identifier(id) => ("IDN".blue(), format!("{:?}",
id).blue()),
260:                     TokenType::Key(num) => ("KEY".yellow(), format!("{}",
num).yellow()),
261:                     TokenType::Error(err) => {
262:                         ("ERR".red().bold(), format!("{:?}", err).red().bold())
263:                     }
264:                     TokenType::SimpleError => ("ERR".red().bold(),
"ERR".red().bold()),
265:                     TokenType::End => ("END".purple().bold(),
"EOF".purple().bold()),
266:                 };
267:                 println!(
268:                     "{} {} {}",
269:                     name,

```

```
270:         format!("{:>2?}-{:>2?}:", x.from, x.to).truecolor(128, 128,
128),
271:         val
272:     );
273:     }
274:     _ => break,
275: }
276: }
277: }
```

Листинг 1 — Код программы

Особенности языка Rust

У языка Rust достаточно много особенностей. Вот основные из них:

1. Сильная типизация – в отличие от Си и C++, Rust сложно обмануть с приведением типов (`void*`) и таким образом «сломать» работу программы.
2. «Строгий» компилятор – применяется принцип *абстракций с нулевой стоимостью*. Идея заключается в том, что достаточно много проверок происходит на этапе компиляции: кто мог воспользоваться переменной, кто мог её «переопределить» и тем самым косвенно «сломать» и т.д. и т.п. из-за чего может «дать по рукам» и не собрать программу, даже если она корректно бы отработала.
3. Функциональный стиль – перенимаются идеи языка Scheme (так написано на википедии, но мне больше нравится сравнение со Scala). Любой блок что-то возвращает, есть сопоставление с образцом (`match`), деструктуризация и некоторые другие «приколы».

Перепечатывать википедию можно долго, но основные тезисы я считаю раскрыл, перейдем к моему мнению об этом языке.

Мнение о языке Rust

Вместо того чтобы отдельно рассмотреть плюсы и минусы, считаю более правильно рассматривать «идеи», которые реализует Rust с разных точек зрения подмечая уже в них как плюсы, так и минусы. Двигаться буду сверху вниз по написанному в этой лабораторной коду:

- «О сборке проекта». Так как хотел добавить подсветку выводимого текста, нужно было добавить библиотеку которая это умеет делать и сделать не просто файл `main.rs`, а «полноценный» проект. Похоже на создание проекта на JavaScript (`package.json` – `Cargo.toml`, `package-lock.json` – `Cargo.lock`), однако намного компактнее и проще – название, версия, зависимости да как `main` называется, намного компактнее и линейнее (в плане формата хранения данных) чем `package.json`. Выглядит удобнее и более продуманно.

- «О `derive`» (6, 17, 24, 32 и 90 строки). Логика понятно – некая стандартная реализация (есть ещё `Clone`, `Copy`, `Hash`). Из плюсов – как по мне, выглядит лучше чем в Java и Kotlin с их `@` из неоткуда, `#` и квадратные скобки, более явно дают понять, что это инструкция для компилятора, да и сама по себе идея что некие стандартные реализации можно добавить канонично в одну строку – выглядит круто, маленький минус что компилятор не добавляет их сам, тот же `Debug` – просто создал новую структуру и не указал: компилятор заругает и предложит дописать, разве не мог он, раз понимает и неявным образом сам дописать? Но он указывает в чем проблема и предлагает её исправить, плюс если бы все таки такие вещи добавлялись в автоматическом режиме, наверное, было бы не так оптимально и страшно что ли (не подходило бы под идеи языка), поэтому – вкусовщина.

- «О enum» (7-15 строки). Enum в типизированном языке, да ещё и не просто набор уникальных чисел-указателей, а скорее оберток для самого разного рода данных – мое почтение. Похоже на глоток свежего воздуха: после TypeScript, кажется странным в типизированных языках отсутствие возможности вернуть «что хочется» (по типу строку или число, число или null, null или null ;)), поэтому однозначный плюс. Очень красиво сочетается с match и функциональными идеями, идеально подходит для данной лабы.

- «О итераторах, доступах и времени жизни» (SmartIterator, 25-88). Если посмотреть на реализацию SmartIterator, то станет видно, что получая на вход строку, он переводится в массив символов и дальше – работа происходит именно с ним. Такая последовательность шагов происходит не случайно и была единственным способом ~~уничтожить Кольцо Всевластья было бросить его в жерло вулкана Ородруин в Мордоре~~ заставить этот кусок кода работать: оставить оригинальную строку и обращаться к ней нельзя, т.к. у строки не реализован доступ по индексу из-за того что символы могут иметь разный размер; оставить итератор по строке тоже не выйдет – изменение строки может его «сломать», поэтому надо указывать время жизни ВСЕГО где окажется этот итератор – то есть в любом другом языке программирования я бы взял у самого себя подписку о том, что обещаю не трогать строку по которой итерируюсь с помощью итератора, то Rust тебе на это говорит известное Станиславовское «Не верю!» и заставляет городить конструкции обозначающие ~~годы~~ время жизни, в которых, если быть честным, я так до конца и не разобрался. Я понимаю почему это происходит: нам не дают лишний раз стрелкнуть себе в ногу, но меньше дискомфорта от этого не становится.\

- «Блоки что-то возвращают» (49-55 и много где ещё). Концепция того что каждый блок что-то возвращает мне понравилась на 12 из 10 – там где я

бы начал городить тернарники и пытаться выносить блоки кода в 2-3 строки, в отдельные функции или вызывать через запятую (как в JS или C/C++) или же создавать деолтное значение, а потом его заполнять (как в том же JS или Go), тут этот код можно сразу вернуть и как сразу все выглядит компактнее, подтянутее и краше. Хотя из-за этого видимо краткая запись тернарного оператора (a?b:c) и отсутствует, а так же далее по коду, видны и некоторые минусы такого подхода – например next_symbol (119-143) – видно что я пишу 3 раза TokenType::SimpleError, хотя мог бы отдельно в if сделать return TokenType::Symbol(ans), а уже после всех проверок TokenType::SimpleError, но мысля только в такой функциональной парадигме о таком забываешь (видимо это чисто я «затупил», надо будет об этом ещё подумать) (листинг 2).

```
119: fn next_symbol(&mut self) -> TokenType {
120:     if Some('\\'') == self.0.next() {
121:         if let Some(ans) = match self.0.next() {
122:             Some('\\'') => None,
123:             Some('\\n') => None,
124:             Some('\\\\') => match self.0.next() {
125:                 Some('n') => Some('\\n'),
126:                 Some(x @ ('\\' | '\\\\')) => Some(x),
127:                 Some(x1) => self.next_number16_4(x1),
128:                 _ => None,
129:             },
130:             x @ _ => x,
131:         } {
132:             if Some('\\'') == self.0.next() {
133:                 return TokenType::Symbol(ans)
134:             }
135:         }
136:     }
137:     TokenType::SimpleError
138: }
```

Листинг 2 — «Исправленный» next_symbol

- «О Optional» (везде, 111-117 особенно). Так как типизация статическая и компилятор всеми правдами и неправдами старается заставить нас писать нормальный код, то нельзя как в Java сказать, что любой объект — это одновременно и он сам или null. Но как же моделировать такое поведение?

Optional. Если все хорошо – это Some со значением, если плохо – то пустой None (ещё раз скажем спасибо enum). Как оказалось при всей своей «математичности» и «сишности» - в Rust можно работать почти также как в JavaScript и много раз вызывать через точку свойства и тем самым изменять (мутировать) данные на ходу. Optional я бы сравнил с Promise, с тем лишь исключением, что Promise – выполнится когда-то там не сразу, а Optional – это именно обертка на результат (повторил кстати такой класс в JavaScript – прикольно получилось). Рассмотрим конкретно на next_number16_4 – функция, которая переводит (если это возможно), следующие 4 символа в итераторе из 16-ричной записи кода символа в сам этот символ. 4 раза подряд «запакуем» соседние Optional в один и в конце попытаемся привести всё к char. Можно записать и другим способом (листинг 3). Основная моя благодарность что такая возможность вообще есть – иначе каждый Optional порождал минимум по дополнительному уровню вложенности и превратился в очень сложно переваримый и нечитаемый кусок кода, а так – чистенько и красивенько.

```
109: fn next_16(a: u32, b: u32) -> u32 { a * 16 + b }
110:
111: fn next_number16_4(&mut self, x: char) -> Option<char> {
112:     to_digit_16(x)
113:     .zip(self.0.next().and_then(to_digit_16)).map(next_16)
114:     .zip(self.0.next().and_then(to_digit_16)).map(next_16)
115:     .zip(self.0.next().and_then(to_digit_16)).map(next_16)
116:     .and_then(char::from_u32)
117: }
```

Листинг 3 — Другой способ сделать next_number16_4

- «О if-let» (104, 121, 141 и т.д.). Очень приятная конструкция, похожая на то что есть в Go – возможность деструктуризации значения и автоматической проверки на соответствие с типом. Очень удобная вещь – если обертка по типу Optional одного типа – то достаём значение и дальше работаем

с ним, если нет – то и доставать нам нечего, так что идем дальше – достаточно удобно, НО нельзя после (или до) этого (а в Go можно) сделать какую-то доп. проверку – например, символ достается и это буква (см. 141-142, 121-132). Правда на эту тему у создателей Rust уже есть issue (<https://github.com/rust-lang/rust/issues/53667>) и при попытке скомпоновать в if let с доп. проверкой, именно туда компилятор нас и отсылает – ждем, надеемся и верим, что когда-нибудь подвезут такую возможность, а пока «увы и ах» – лишний уровень вложенности: не критично, но обидно.

- «Остатки». Match и Loop – прикольные блоки: и значение возвращают, и возможностей очень много. В main пришлось отказать от идеи по-честному вызвать итератор через for, так как иначе не было бы доступа к HashMap идентификаторов: обидно, но логично – управление, то в for надо передать, не получилось «и рыбку съесть и на качелях покачаться». Не const, а mut, не NotNull, а Optional – механизм доступов и разрешений перевернут в более логичную сторону: чтобы делать дополнительные финты, надо именно «запросить» это, написать доп. слово, а не наоборот – доп. словами «ограничивать» логику.

Вывод

В ходе лабораторной работы было приобретен навык разработки на языке Rust. Язык, конечно, строговат, но благодаря многочисленным особенностям – писать на нем очень приятно. Хочу теперь написать свой язык с блоками возвращающие значения, mut и Optional.

Ссылка на репозиторий:

<https://github.com/ZhukDmitryOlegovich/compiler-labs-lab4>